SAMSKRUTI COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to JNTUH.)

Kondapur (V), Ghatkesar (M), Medchal (Dist)

Course Hand Out

Subject Name: Operating Systems

Prepared by: G. Radha Devi, Assistant Professor, CSE

Year, Semester, Regulation: II Year- II Sem (R18)

UNIT-1

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware
Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure
Computer system can be divided into four components

- Hardware – provides basic computing resources
  CPU, memory, I/O devices

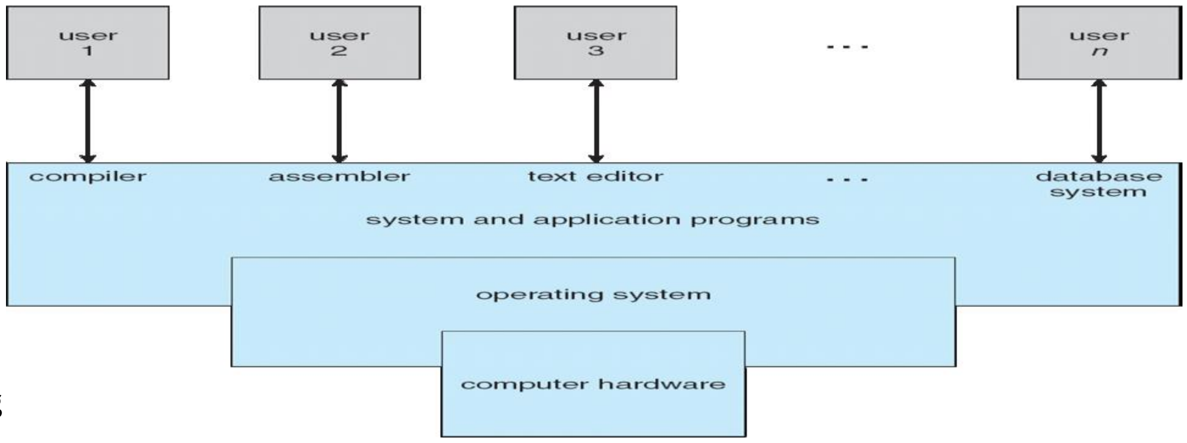
  Operating system

  Controls and coordinates use of hardware among various applications and users

- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  Word processors, compilers, web browsers, database systems, video games

- Users
  People, machines, other computers

Four Components of a Computer System



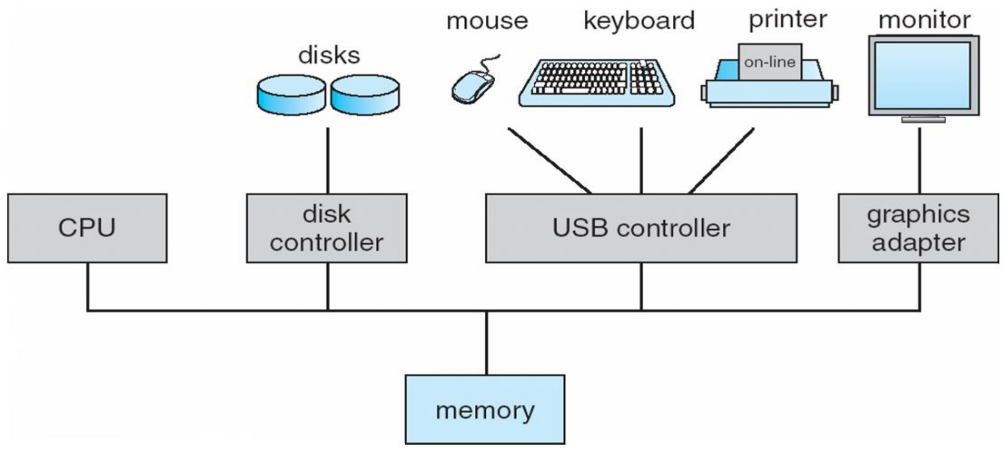
Operating

- OS is a **resource allocator**
- Manages all resources
- Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
- Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- Everything a vendor ships when you order an operating system" is good approximation but varies wildly.

- "The one program running at all times on the computer" is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program.

Computer System Organization

- Computer-system operation
- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles



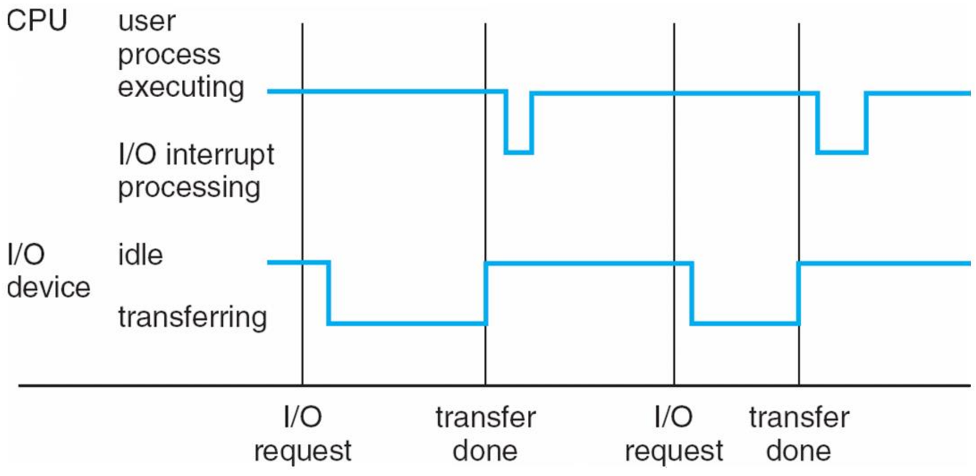
Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing An *interrupt*

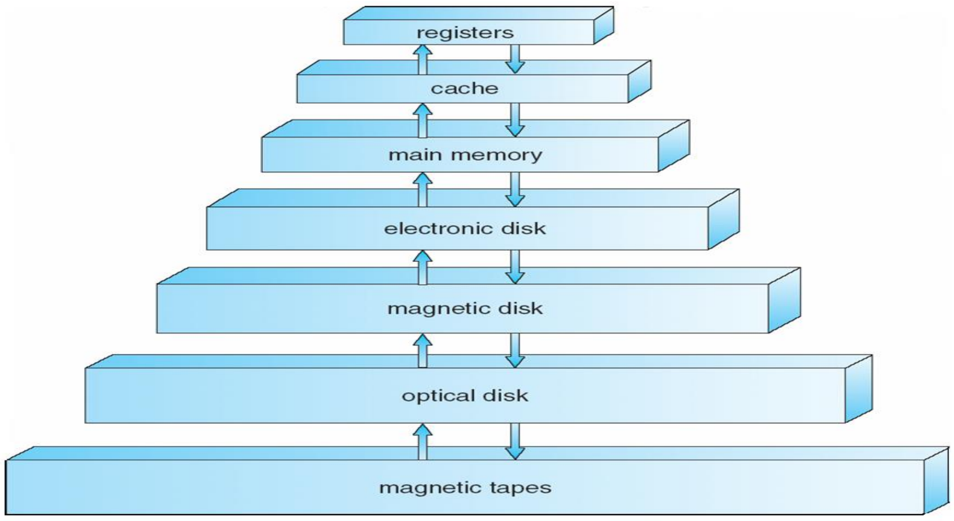
Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- **polling**
- **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



can be viewed as a last *cache* for secondary storage



Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
- If it is, information used directly from the cache (fast)
- If not, data copied to cache and used there
- Cache smaller than storage being cached
- Cache management important design problem
- Cache size and replacement policy

Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
- Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
- Also known as parallel systems, tightly-coupled systems

Advantages include

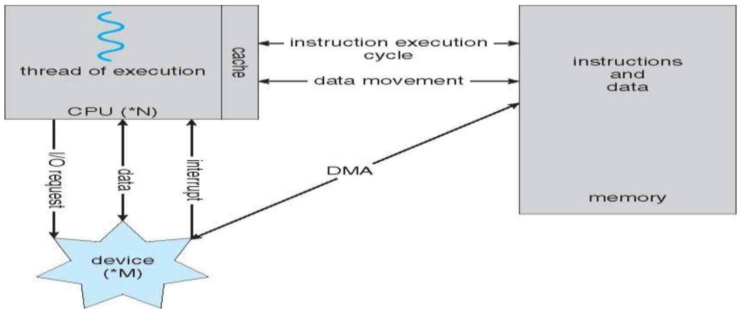
1. Increased throughput 2.Economy of scale

3.Increased reliability – graceful degradation or fault tolerance

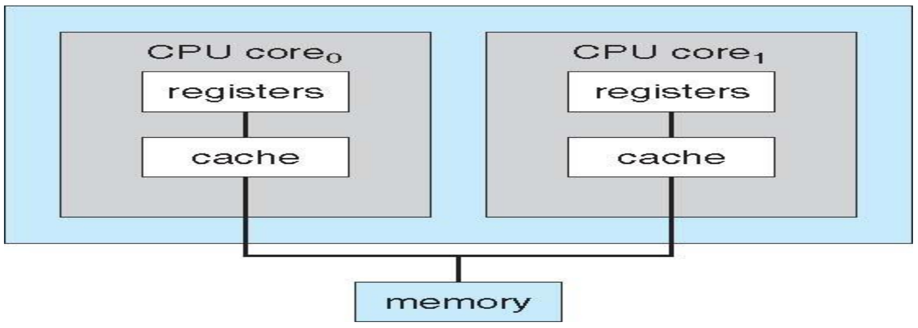
Two types 1.Asymmetric Multiprocessing 2.Symmetric Multiprocessing

How a Modern Computer Works

Symmetric Multiprocessing Architecture



A Dual-Core Design



Clustered Systems

- Like multiprocessor
- systems, but multiple systems working together
- Usually sharing storage via a storage-area network (SAN)
- Provides a high-availability service which survives failures
  Asymmetric clustering has one machine in hot-standby mode

  Symmetric clustering has multiple nodes running applications, monitoring each other

- Some clusters are for high-performance computing (HPC)
  Applications must be written to use parallelization

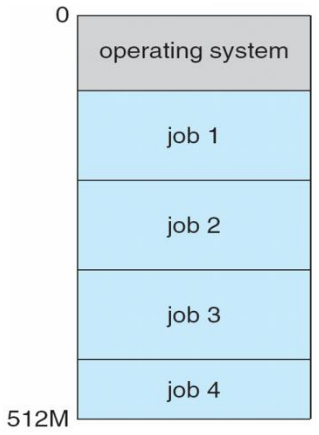
Operating System Structure

- **Multiprogramming** needed for efficiency
- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to Execute
- A subset of total jobs in system is kept in memory
- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
- **Response time** should be < 1 second

- Each user has at least one program executing in memory [**process**
- If several jobs ready to run at the same time [**CPU scheduling**
- If processes don’t fit in memory, **swapping** moves them in and out to run

Virtual memory allows execution of processes not completely in memory

Memory Layout for Multiprogramming System



Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
- Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each Other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
- **User mode** and **kernel mode**
- **Mode bit** provided by hardware

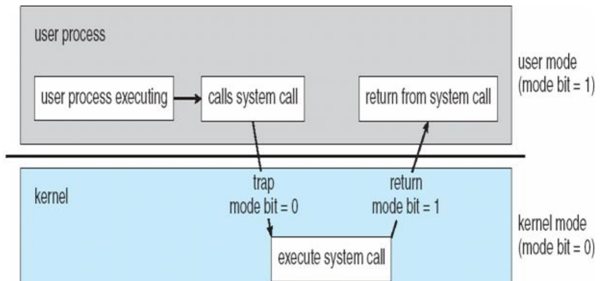
Provides ability to distinguish when system is running user code or kernel code

Some instructions designated as **privileged**, only executable in kernel mode

System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

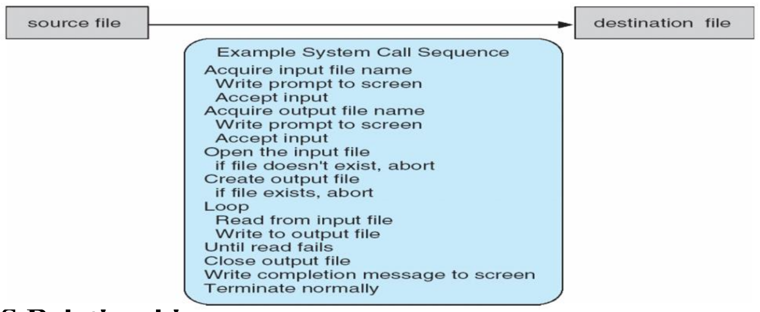
- Timer to prevent infinite loop / process hogging resources
- Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



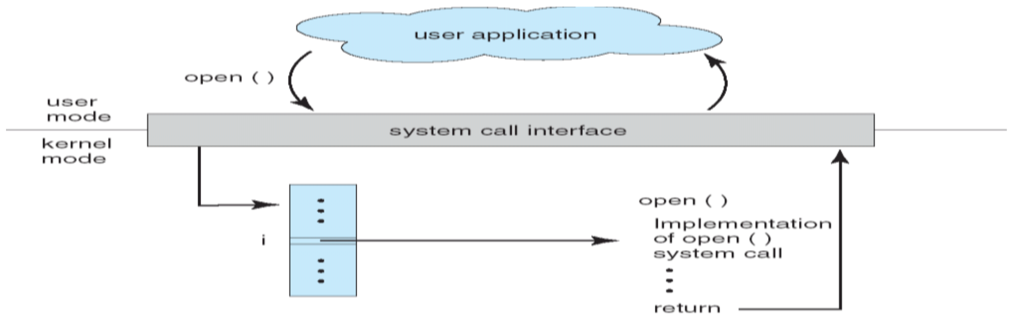
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call usenThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

Example of System Calls



API – System Call – OS Relationship



<u>IMPORTANT QUESTIONS</u>:

1. Define OS, Explain Various Functions of OS.

2. Explain Different Types of OS.

3. What Is System Call? Discuss Various Types of System Calls.

4. Explain OS Structure.

5. Write Short Notes on Virtual Machines.

UNIT-II

Process Concepts

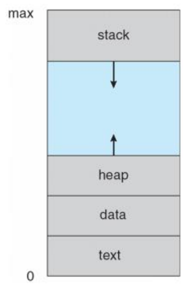
- An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

A process includes:

- program counter
- stack
- data section

Process in Memory

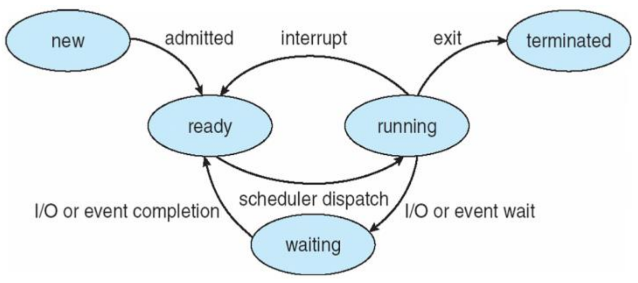


Process State

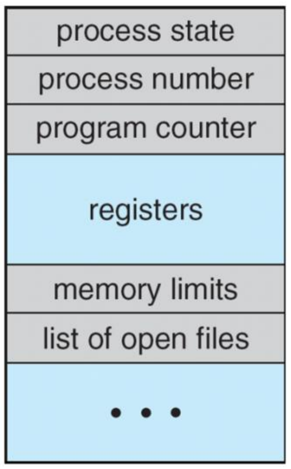
As a process executes, it changes *stat*e

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Diagram of Process State



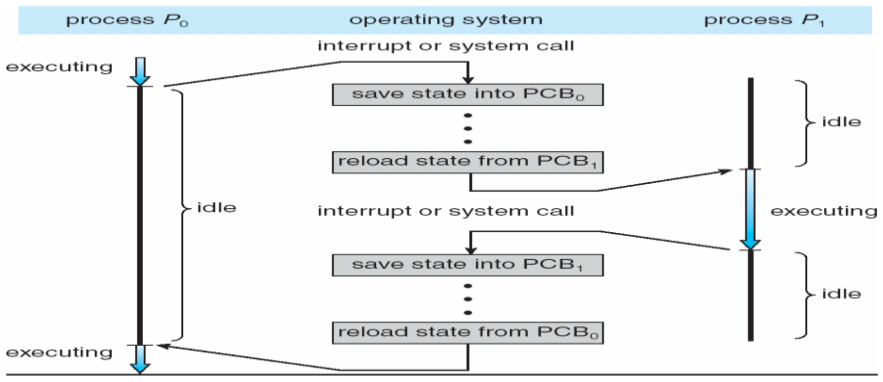
Process Control Block (PCB)



Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

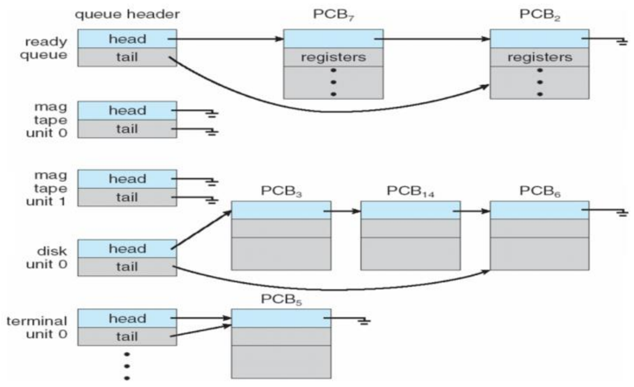
CPU Switch from Process to Process



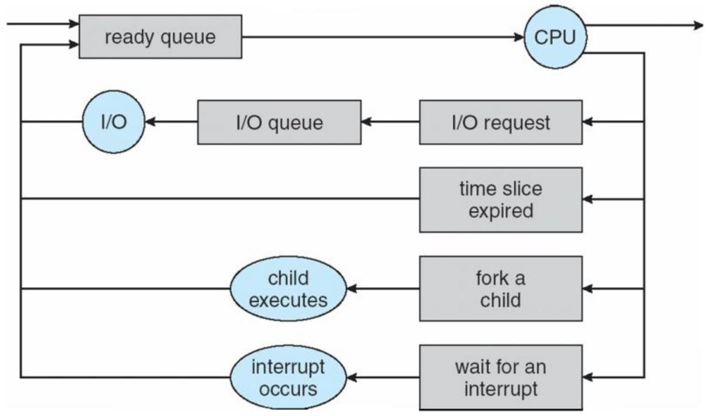
Process Scheduling Queues

Job queue – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



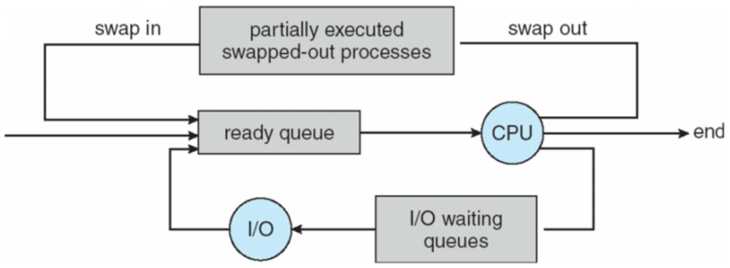
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



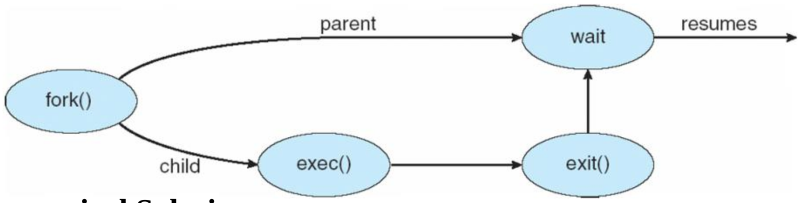
- Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*

- Processes can be described as either:
- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

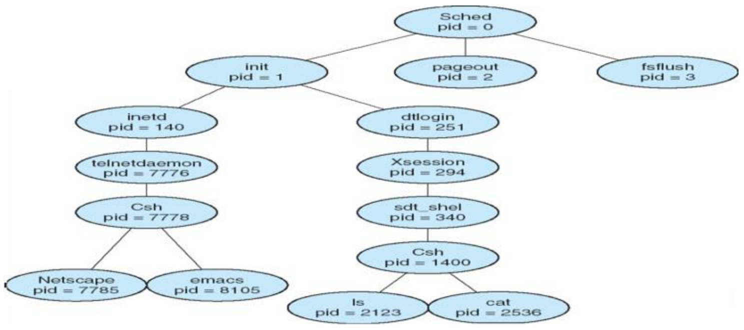
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources
- Child has a program loaded into it
- UNIX examples
- **fork** system call creates new process
- **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation



A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
- Output data from child to parent (via **wait**)
- Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
- Child has exceeded allocated resources
- Task assigned to child is no longer required
- If parent is exiting

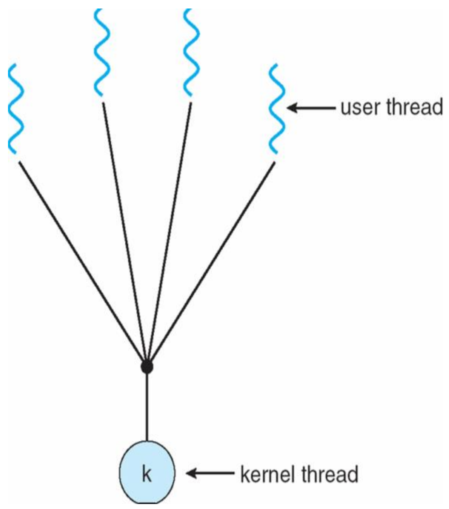
Some operating system do not allow child to continue if its parent terminates

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

Many user-level threads mapped to single kernel thread



Examples:

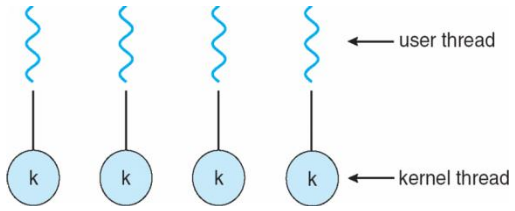
- Solaris Green Threads
- GNU Portable Threads

One-to-One

Each user-level thread maps to kernel thread

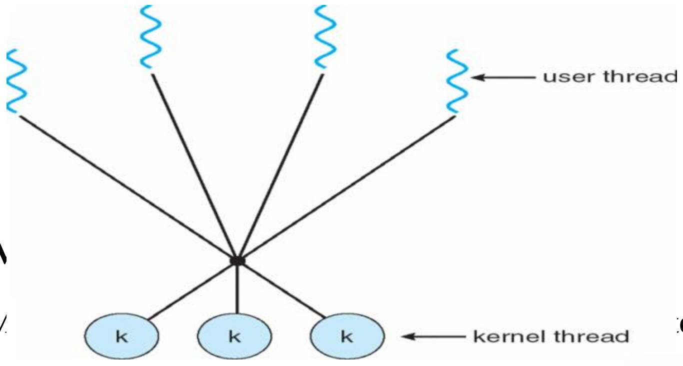
Examples Windows NT/XP/2000

Linux Solaris 9 and later



Many-to-Many Model

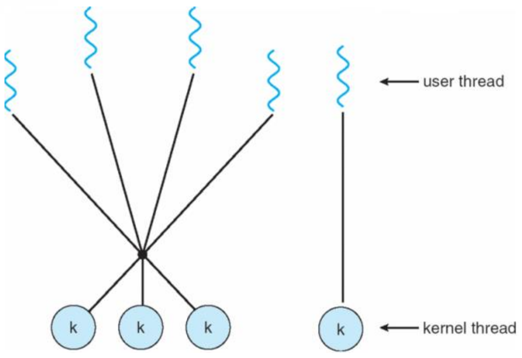
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
  Windows NT/2000 with the *Thread Fiber* package



Two-level M

Similar to M o kernel thread

Examples



- IRIX
- HP-UX
- Tru64 UNIX

- Solaris 8 and earlier

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
- Library entirely in user space
- Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library

Common in UNIX operating systems (Solaris, Linux, Mac OS X)

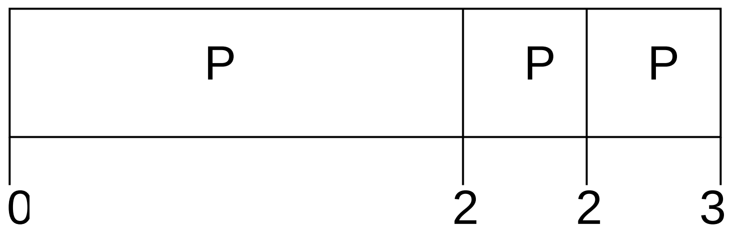
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

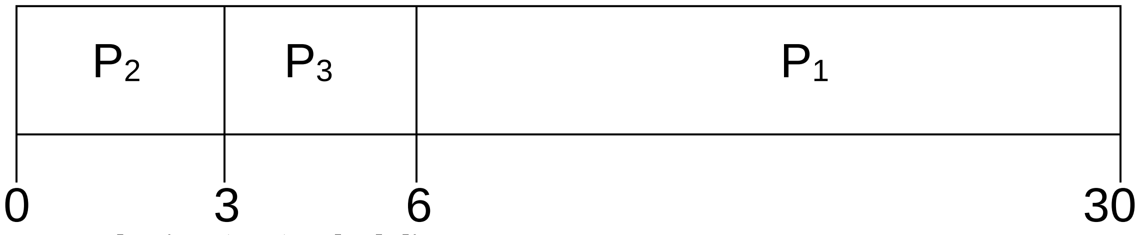
Suppose that the processes arrive in the order

P_2 , P_3 , P_1

The Gantt chart for the schedule is:nnnn Waiting time for P_1 = 6; P_2 = 0; P_3 = 3nAverage waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

Convoy effect short process behind long process



Shortest-Job-First (SJF) Scheduling

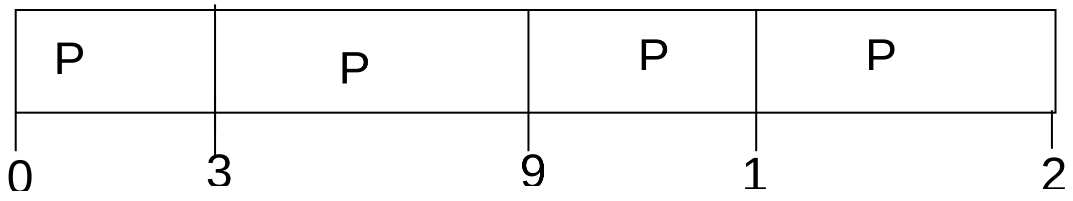
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

SJF scheduling chart

average waiting time = (3 + 16 + 9 + 0) / 4 = 7the length of the next CPU request



Process Synchronization

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Assume that each process executes at a nonzero speed

No assumption concerning relative speed of the N processes

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
- int turn;
- Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
- Also known as mutex locksn Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion Semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
- Could now have busy waiting in critical section implementation

But implementation code is short

Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
- value (of type integer)
- pointer to next record in the list
- Two operations:
- block – place the process invoking the operation on the appropriate waiting queue.
- wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

- The structure of the producer process

Readers-Writers Problem

A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do **not** perform any updates

- Writers – can both read and written Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
- Data set
- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer read count initialized to 0

Dining-Philosophers Problem

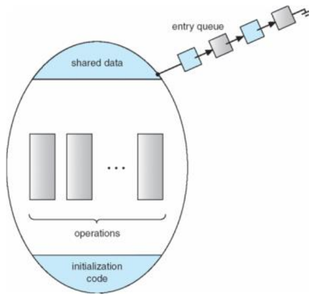
- Shared data
- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Only one process may be active within the monitor at a time

Schematic view of a Monitor



Condition Variables

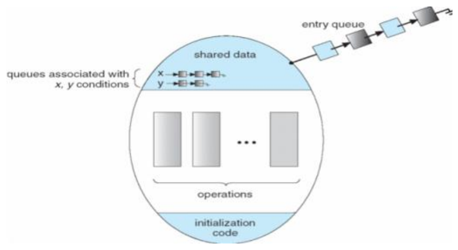
condition x, y;

Two operations on a condition variable:

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



IMPORTANT QUESTIONS:

1) Define Process .Explain about Process States with Diagram.

2) What Is Scheduling. Explain About Types of Schedulers.

3) Explain About Various CPU Scheduling Algorithms.

4) Write Short Notes on A) Multilevel Scheduling

B) Multiprocessor Scheduling
C)Time Scheduling

5) Explain About Process Synchronization and Critical Section Problem.

6) Solutions for Critical Section Problem

A) Petersons Solution B) Semaphore C) Monitors

7) Discuss Classical Synchronization Problems

A) Readers Writers Problem B) Producer Consumer Problem

C) Dining Philosophers Problem

UNIT-III

Deadlocks

To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks? To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Example: System has 2 disk drives

P_1 and P_2 each hold one disk drive and each needs another one

Example: semaphores *A* and *B*, initialized to 1

P_0 P

```
wait (A);        wait(B)

wait (B);        wait(A)
```

System Model

Resource types R_1, R_2, . . ., R_m.*CPU cycles, memory space, I/O devices*

Each resource type R_i has W_i instances.Each process utilizes a resource as follows:

request

use

release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

Mutual exclusion: only one process at a time can use a resource

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

Circular wait: there exists a set {P_0, P_1, …, P_0} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Deadlock Prevention

Restrain the ways request can be made

Mutual Exclusion – not required for sharable resources; must hold for non sharable resources

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources. Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none. Low resource utilization; starvation possible

Deadlock Avoidance

Requires that the system has some additional *a priori* information available. Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j.The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

Multiple instances Each process must a priori claim maximum use. When a process requests a resource it may have to wait .When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types. N **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available **Max***:* n *x* m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j **Allocation***:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_jn**Need***:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.*Need* [i,j] = *Max*[i,j] – *Allocation* [i,j]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:*Work* = *Available*

Finish [i] = *false* for i = 0, 1, …, n- 1

2. Find and i such that both: (a) *Finish* [i] = *false*(b) $Need_i$ £ *Work* If no such i exists, go to step 4

3. *Work* = *Work* + $Allocation_i$ *Finish*[i] = *true* go to step 2

4. If *Finish* [i] == true for all i, then the system is in a safe state

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:(a) *Work* = *Available*(b) For *i* = 1,2, …, *n*, if $Allocation_i$ ¹ 0, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*2. Find an index *i* such that both:(a) *Finish*[*i*] == *false*(b) $Request_i$ £ *Work*If no such *i* exists, go to step 4

3. *Work* = *Work* + $Allocation_i$ *Finish*[*i*] = *true* go to step 24. If *Finish*[*i*] == false, for some *i*, 1 £ *i* £ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances) Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4**etection-**

Algorithm Usage

When, and how often, to invoke depends on: How often a deadlock is likely to occur?

How many processes will need to be rolled back?

one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock

Protection

Discuss the goals and principles of protection in a modern computer system Explain how protection domains combined with an access matrix are used to specify the resources a process may access Examine capability and language-based protection systems

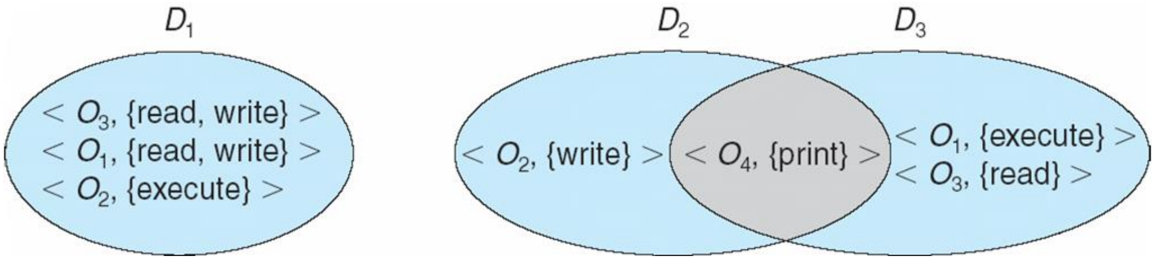
Goals of Protection: Operating system consists of a collection of objects, hardware or software Each object has a unique name and can be accessed through a well-defined set of operations Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.

Principles of Protection

Guiding principle – principle of least privilege Programs, users and systems should be given just enough privileges to perform their tasks

Domain Structure

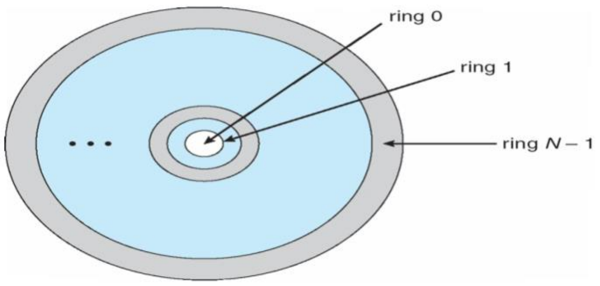
Access-right = <*object-name*, *rights-set*> where *rights-set* is a subset of all valid operations that can be performed on the object. Domain = set of access-rights



System consists of 2 domains: User Supervisor UNIX Domain = user-id Domain switch accomplished via file system

Each file has associated with it a domain bit (setuid bit)

When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset



Domain Implementation (MULTICS)

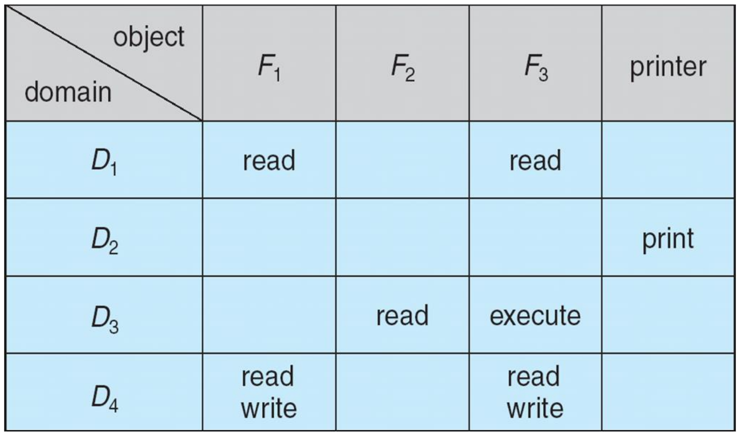
Let D_i and D_j be any two domain rings

If $j < I$ Þ D_i Í D_j

Access Matrix

View protection as a matrix (*access matrix*) Rows represent domains Columns represent objects

Access(i, j) is the set of operations that a process executing in Domain$_i$ can invoke on Object$_j$



object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Implementation of Access Matrix

Each column = Access-control list for one object
Defines who can perform what operation.

Domain 1 = Read, Write
Domain 2 = Read
Domain 3 = Read

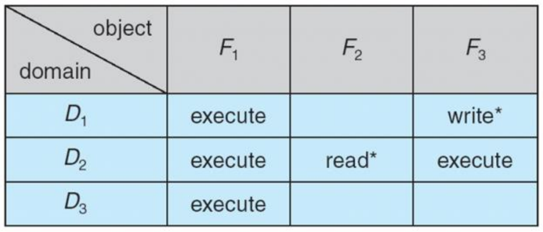
Each Row = Capability List (like a key) Fore each domain, what operations allowed on what objects.

Object 1 – Read

Object 4 – Read, Write, Execute

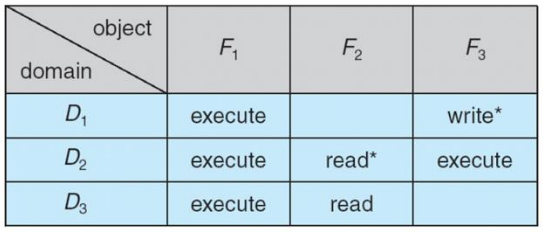
Object 5 – Read, Write, Delete, Copy

Access Matrix with *Copy* Rights



object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

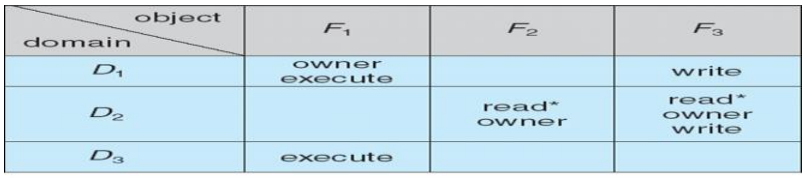


object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

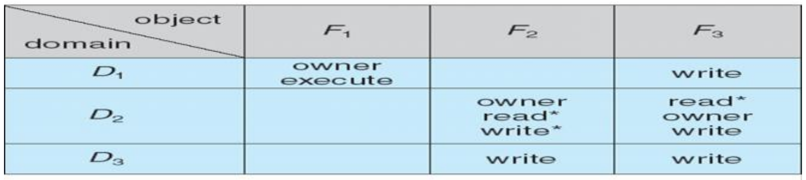
Access Matrix with *Owner* Rights

Modified Access Matrix of Figure B



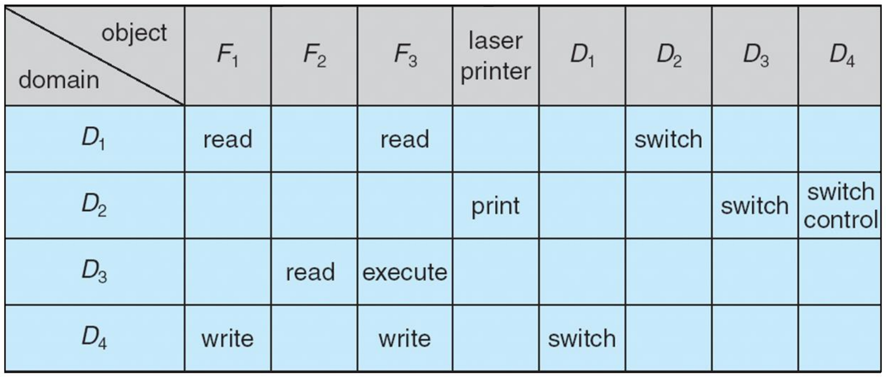
object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)



object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Revocation of Access Rights

Access List – Delete access rights from access list Simple and Immediate

Capability List – Scheme required to locate capability in the system before capability can be revoked Reacquisition Back-pointers Indirection Keys

Capability-Based Systems

Hydra Fixed set of access rights known to and interpreted by the system Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights Cambridge CAP System Data capability - provides standard read, write, execute of individual storage segments associated with object Software capability -interpretation left to the subsystem, through its protected procedures.

Language-Based Protection

Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

IMPORTANT QUESTIONS:

1) What Is A Dead Lock?
2) Explain Characteristics Of Deadlock.
3) Explain About A) Deadlock Prevention
   B) Deadlock Detection
   C) Dead Lock Avoidance
   D) Deadlock Recovery

4) Explain Bankers Algorithm.

5) Explain How Protection Is Implemented For Access Matrix.

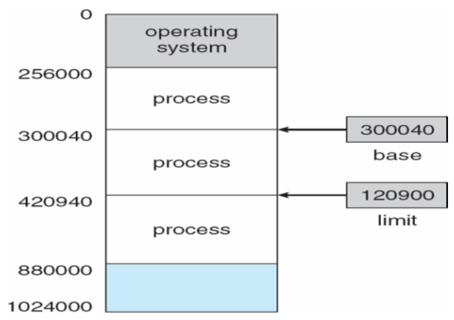
UNIT-IV

Memory Management

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

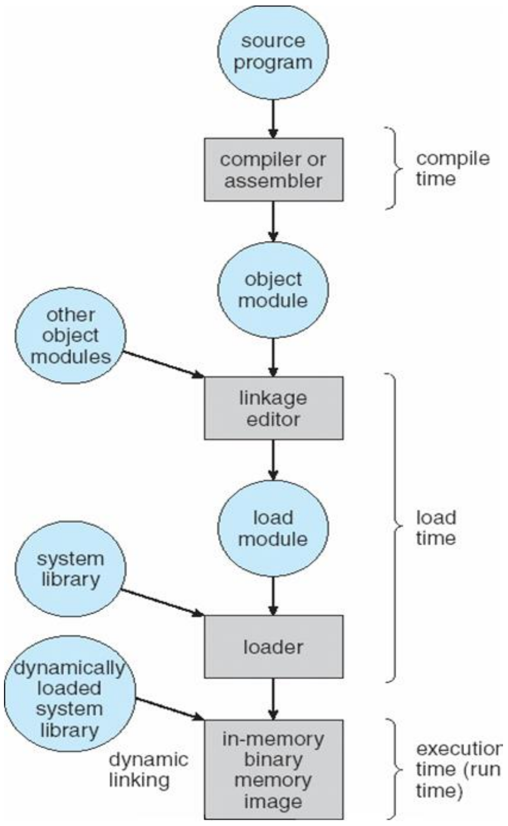
A pair of **base** and **limit** registers define the logical address space



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

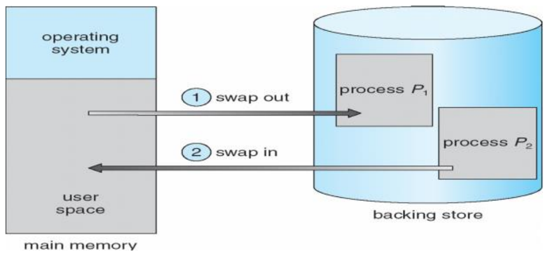
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Contiguous Allocation

- Main memory usually into two partitions:
- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- Base register contains value of smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*

Dynamic Storage-Allocation Problem

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
- Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block

- Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - I/O problem

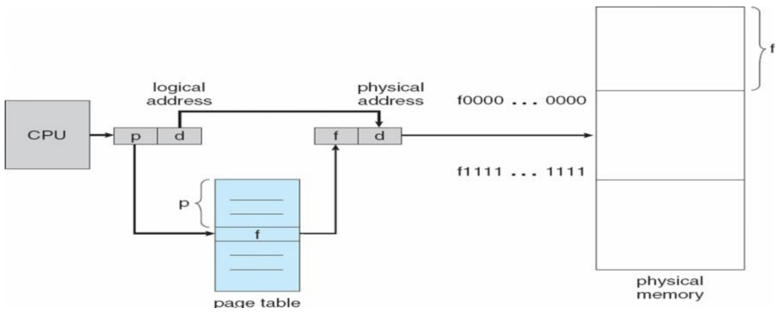
    Latch job in memory while it is involved in I/O

    Do I/O only into OS buffers

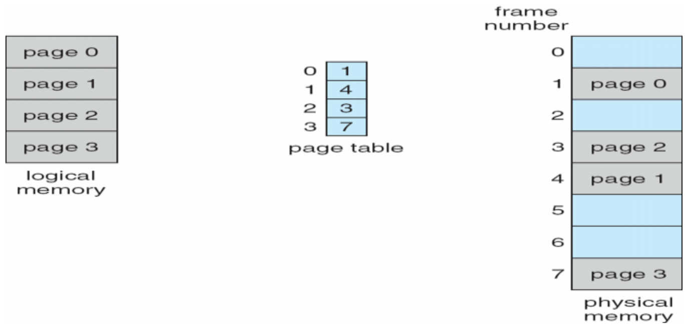
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called pagesnKeep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation
-

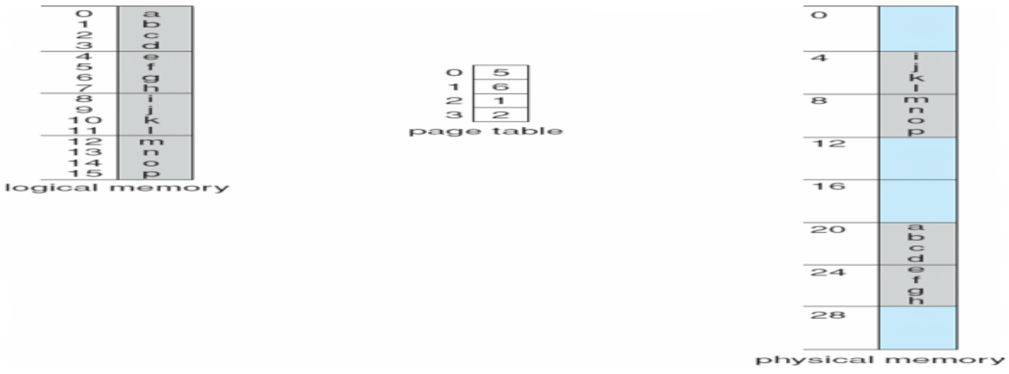
Paging Hardware



Paging Model of Logical and Physical Memory

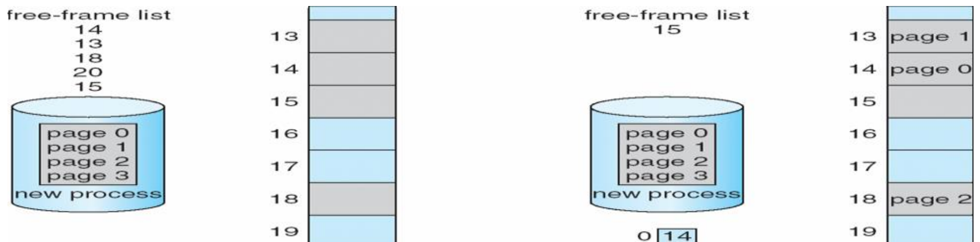


Paging Example



32-byte memory and 4-byte pages

Free Frames



Implementation of Page Table

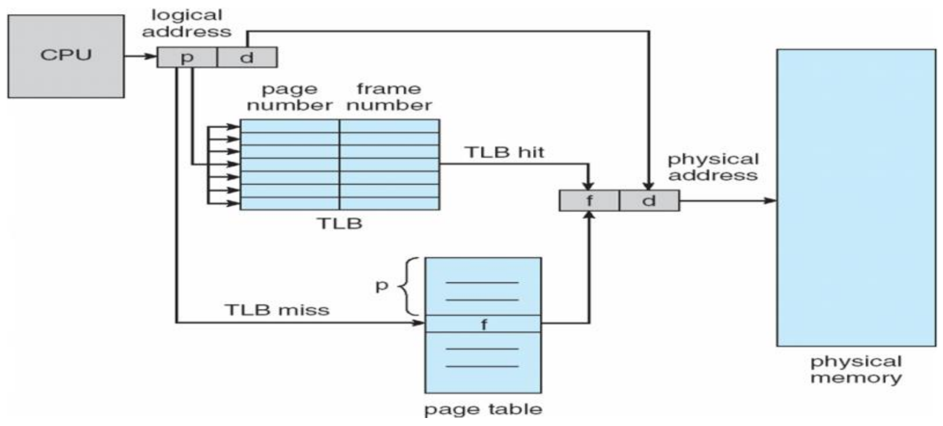
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

- Associative memory – parallel search
- Address translation (p, d)
- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Page #	Frame #

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = e time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = an **Effective Access Time** (EAT)

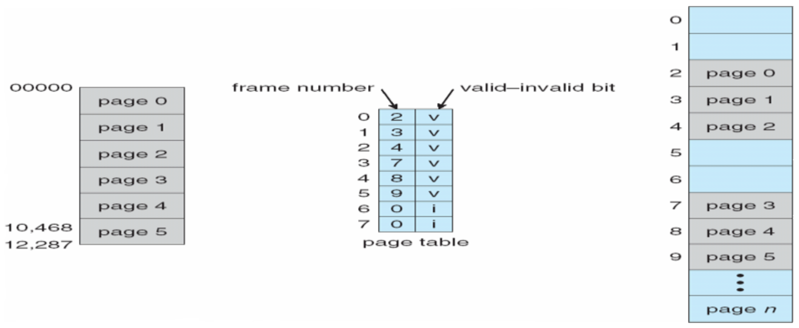
$$EAT = (1 + e)\,a + (2 + e)(1 - a)$$

$$= 2 + e - a$$

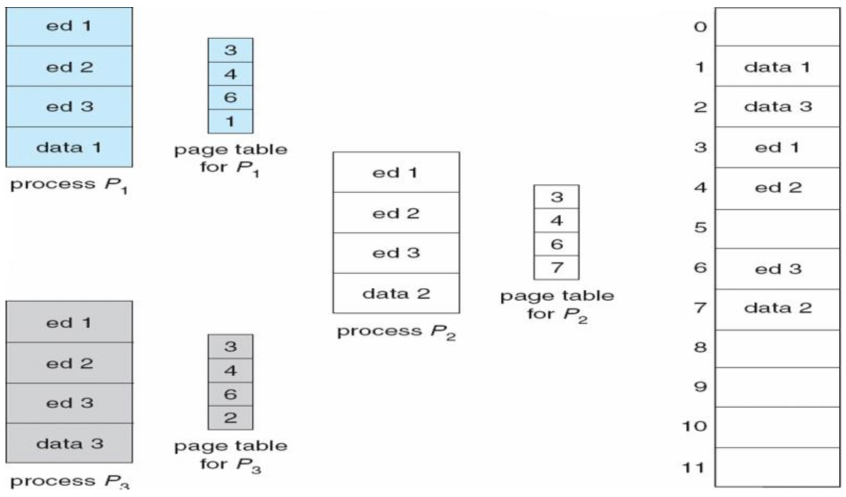
Memory Protection

- Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space
- **Valid (v) or Invalid (i) Bit In A Page Table**



Shared Pages Example



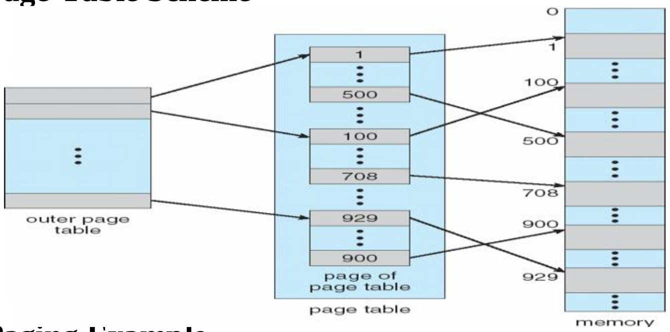
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

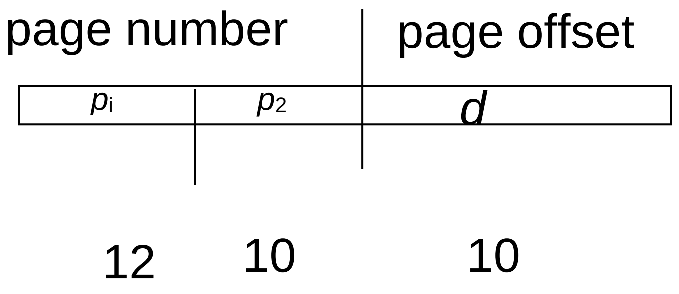
Two-Level Page-Table Scheme



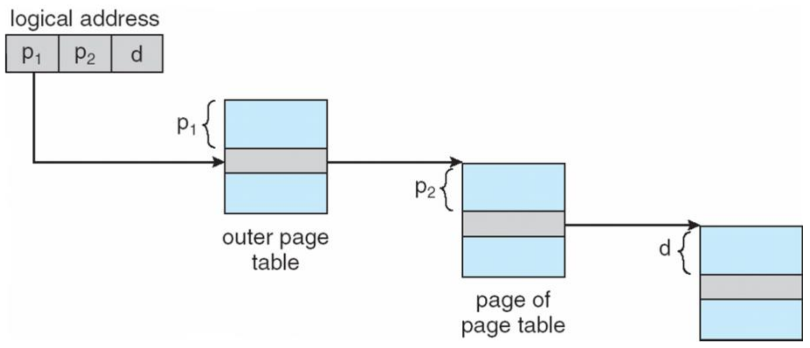
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
- a page number consisting of 22 bits
- a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
- a 12-bit page number
- a 10-bit page offset
- Thus, a logical address is as follows:

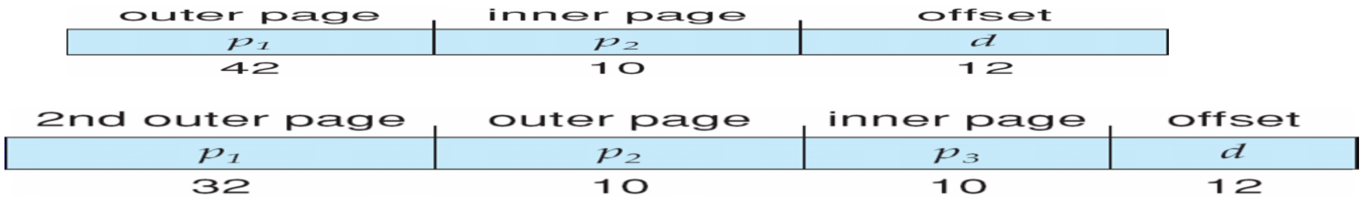
  where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table



Address-Translation Scheme

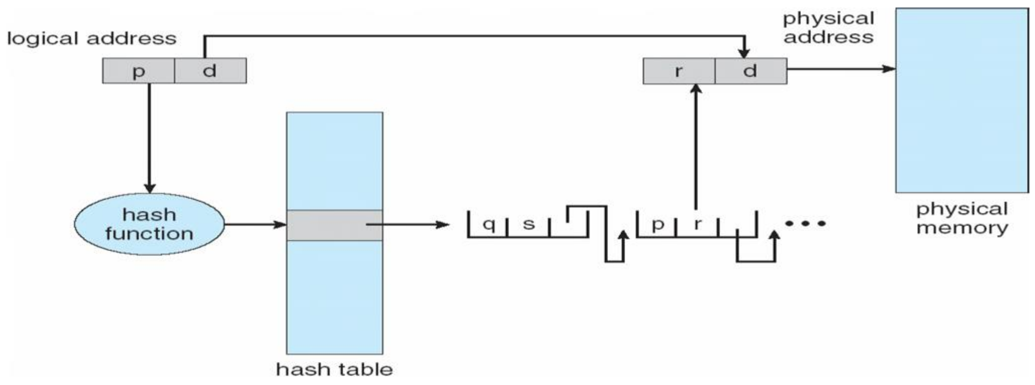


Three-level Paging Scheme



Hashed Page Tables

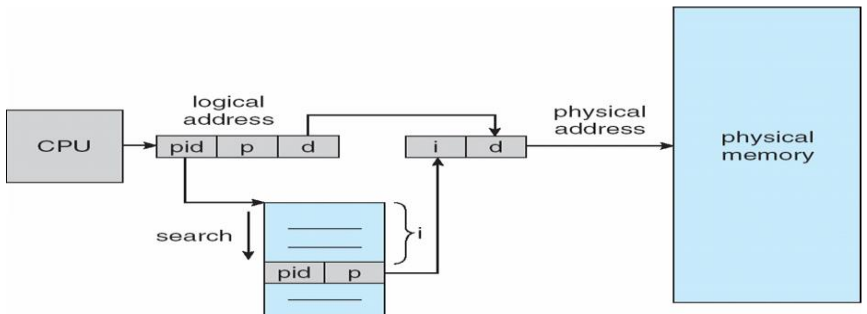
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted



Hashed Page Table Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

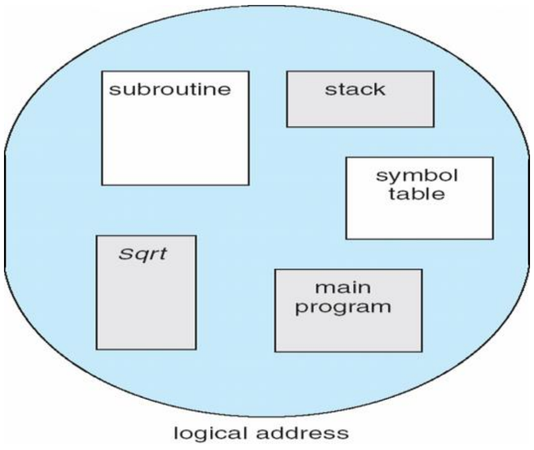
Inverted Page Table Architecture



Segmentation

Memory-management scheme that supports user view of memory

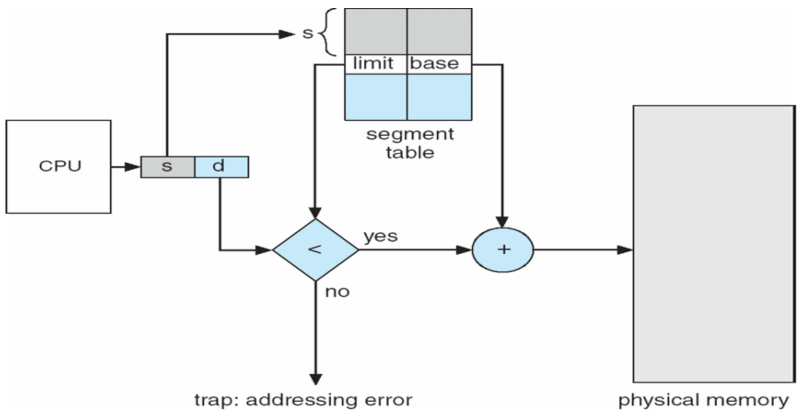
- A program is a collection of segments
- A segment is a logical unit such as:
- main program
- procedure function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays



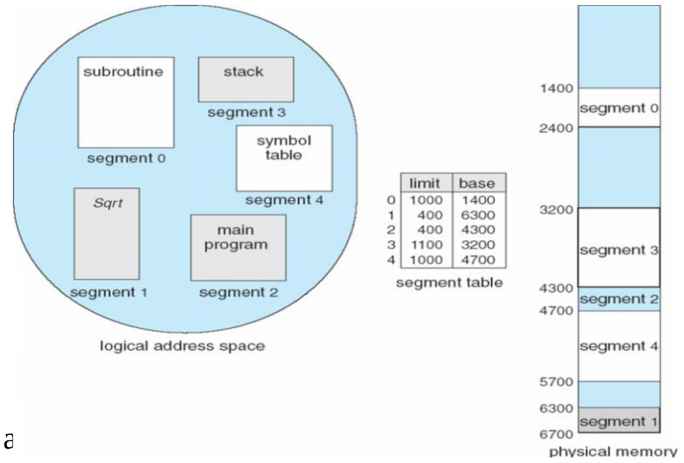
Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program; segment number ***s*** is legal if ***s*** < **STLR**
- Protection
- With each entry in segment table associate:
- validation bit = 0 Þ illegal segment
- read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



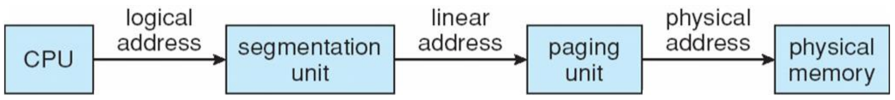
Example of Segmentation

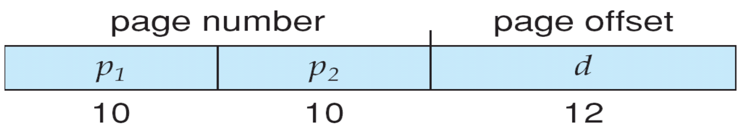


Example: The Intel Pentium

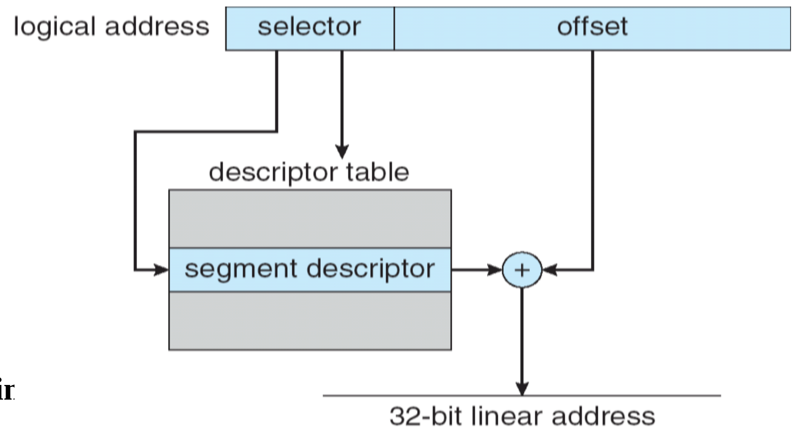
- Supports both segmentation a
- CPU generates logical address
- Given to segmentation unit Which produces linear addresses
- Linear address given to paging unit Which generates physical address in main memory Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium

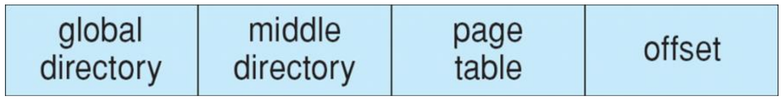




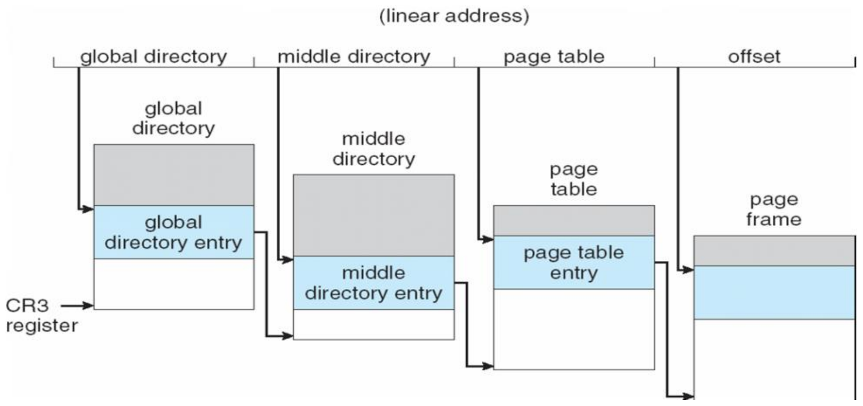
Intel Pentium Segmentation



Linear Address ir



Three-level Paging in Linux



IMPORTANT QUESTIONS:

1) Define File ,Explain File Access Methods.
2) Write Short Notes File System Mounting ,File Sharing, Protection file.
3) Explain About File Allocation Methods.
4) Discuss About Free Space Management.
5) Explain About Directory Structure And Directory Implementation.
6) Explain About Disk Scheduling Algorithms.
7) Explain About Swap Space Management.

UNIT-V

File system Interface

To explain the function of file systems To describe the interfaces to file systems To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures To explore file-system protection.

File Concept

Contiguous logical address space

Types:

Data, numeric, character, binary Program

File Structure

None - sequence of words, bytes Simple record structure Lines Fixed length Variable length Complex Structures Formatted document Relocatable load file Can simulate last two with first method by inserting appropriate control characters Who decides: Operating system, Program.

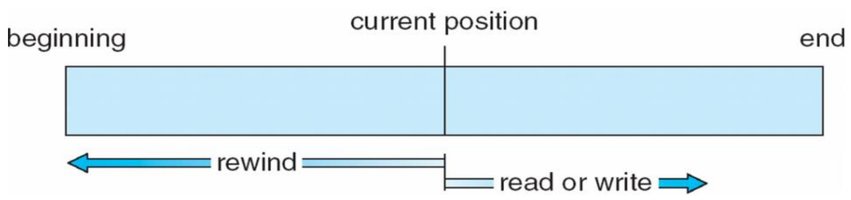
Access Methods

Sequential Access read next, write next ,reset , no read after last write,   (rewrite)

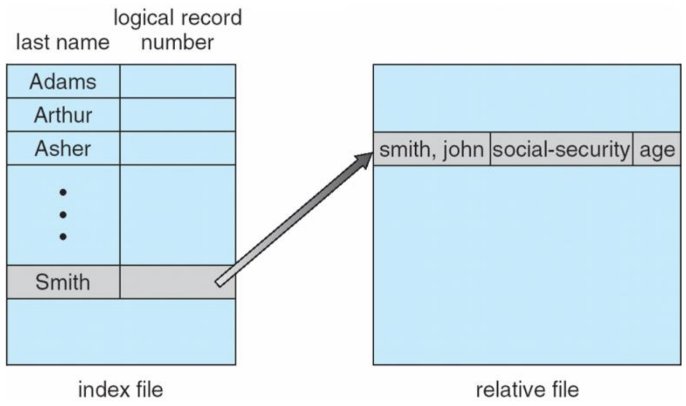
Direct Access read *n,* write *n,* position to *n,* read next,write next ,rewrite *n.*

n = relative block number

Sequential-access File



Example of Index and Relative Files



Directory Structure

A collection of nodes containing information about all files Both the directory structure and the files reside on disk Backups of these two structures are kept on tapes

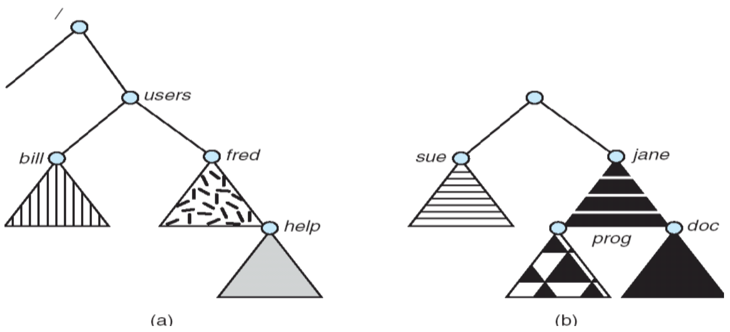
Disk Structure

Disk can be subdivided into partitions Disks or partitions can be RAID protected against failure. Disk or partition can be used raw – without a file system, or formatted with a file system Partitions also known as minidisks, slices Entity containing file system known as a volume Each volume containing file system also tracks that file system's info in device directory or volume table of contents As well as general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer.

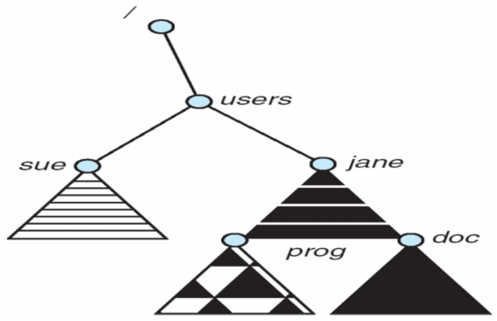
File System Mounting

A file system must be **mounted** before it can be accessed A unmounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point.**

(a) Existing. (b) Unmounted Partition



Mount Point



File Sharing

Sharing of files on multi-user systems is desirablenSharing may be done through a **protection** scheme On distributed systems, files may be shared across a networknNetwork File System (NFS) is a common distributed file-sharing method

File Sharing – Multiple Users

User IDs identify users, allowing permissions and protections to be per-user **Group IDs** allow users to be in groups, permitting group access rights.

File Sharing – Remote File Systems

Uses networking to allow file system access between systems Manually via programs like FTP Automatically, seamlessly using **distributed file systems** Semi automatically via the **world wide web**

Protection

File owner/creator should be able to control

l**Read** l**Write** l**Execute** l**Append** l**Delete** l**List**

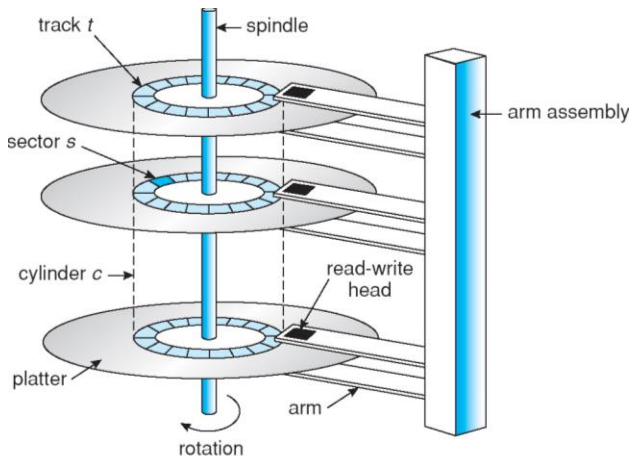
Mass-storage structure

Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers Drives rotate at 60 to 200 times per second Transfer rate is rate at which data flow between drive and computer Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency) Head crash results from disk head making contact with the disk surface

That's bad Disks can be removable Drive attached to computer via I/O bus Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI Host controller in computer uses bus to talk to disk controller built into drive or storage array

Moving-head Disk Mechanism



Magnetic tape Was early secondary-storage medium Relatively permanent and holds large quantities of data Access time slow Random access ~1000 times slower than disk Mainly used for backup, storage of infrequently-used data, transfer medium between systems Kept in spool and wound or rewound past read-write head Once data under head, transfer rates comparable to disk 20-200GB typical storage Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT.

Disk Structure

Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially Sector 0 is the first sector of the first track on the outermost cylinder Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Attachment

Host-attached storage accessed through I/O ports talking to I/O busses SCSI itself is a bus, up to 16 devices on one cable, SCSI initiator requests operation and SCSI targets perform tasks Each target can have up to 8 logical units (disks attached to device controller FC is high-speed serial architecture Can be switched fabric with 24-bit address space – the basis of storage area networks (SANs) in which many hosts attach to many storage units Can be arbitrated loop (FC-AL) of 126 devices.

Disk Scheduling

The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth Access time has two major components Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head Minimize seek time Seek time » seek distance Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer Several algorithms exist to schedule the servicing of disk I/O requests . We illustrate them with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67   Head pointer 53

Swap-Space Management

Swap-space — Virtual memory uses disk space as an extension of main memory Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition Swap-space management l4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment Kernel uses swap maps to track swap-space use Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.

IMPORTANT QUESTIONS:

1) Differences between Logical& Physical Address Space.
2) Explain About Contiguous Memory Allocation And Memory Allocation Algorithms.
3) Explain About Paging And Page Table Structures.
4) Explain About Segmentation.
5) Define Fragmentation And Discuss Differences Between Internal And External Fragmentation.
6) Discuss about Virtual Memory.
7) Explain about Demand Paging.

8) Explain about Various Page Replacement Algorithms.

9) Define Thrashing. Explain different Thrashing Control Methods.

10) Discuss Different Frame Allocation Methods.